US 20020104076A1

(19) United States

(12) Patent Application Publication (10) Pub. No.: US 2002/0104076 A1

SHAYLOR (43) Pub. Date: Aug. 1, 2002

(54) CODE GENERATION FOR A BYTECODE COMPILER

(76) Inventor: NIK SHAYLOR, NEWARK, CA (US)

Correspondence Address:
HOGAN & HARTSON LLP
IP GROUP, COLUMBIA SQUARE
555 THIRTEENTH STREET, N.W.
WASHINGTON, DC 20004 (US)

(*) Notice: This is a publication of a continued prosecution application (CPA) filed under 37 CFR 1.53(d).

(21) Appl. No.: 09/108,061

(22) Filed: Jun. 30, 1998

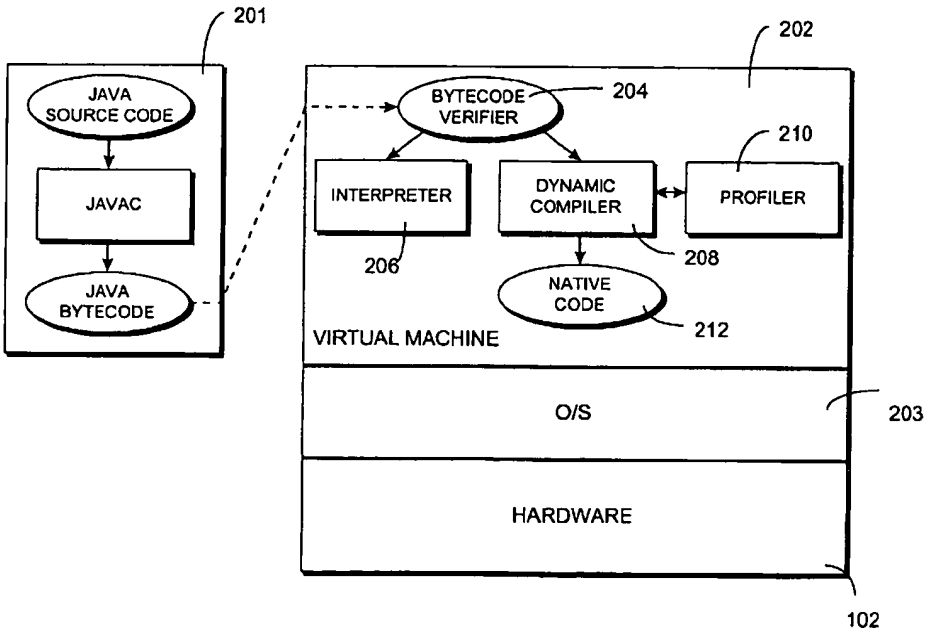
Publication Classification

(51) Int. Cl.[7] G06F 9/45

(52) U.S. Cl. 717/148; 717/162

(57) ABSTRACT

A method, system and apparatus for generating and optimizing native code in a runtime compiler from a group of bytecodes presented to the compiler. The compiler accesses information that indicates a likelihood that a class will be a particular type when accessed by the running program. Using the accessed information, the compiler selects a code generation method from a plurality of code generation methods. A code generator generates optimized native code according to the selected code generation method and stores the optimized native code in a code cache for reuse.



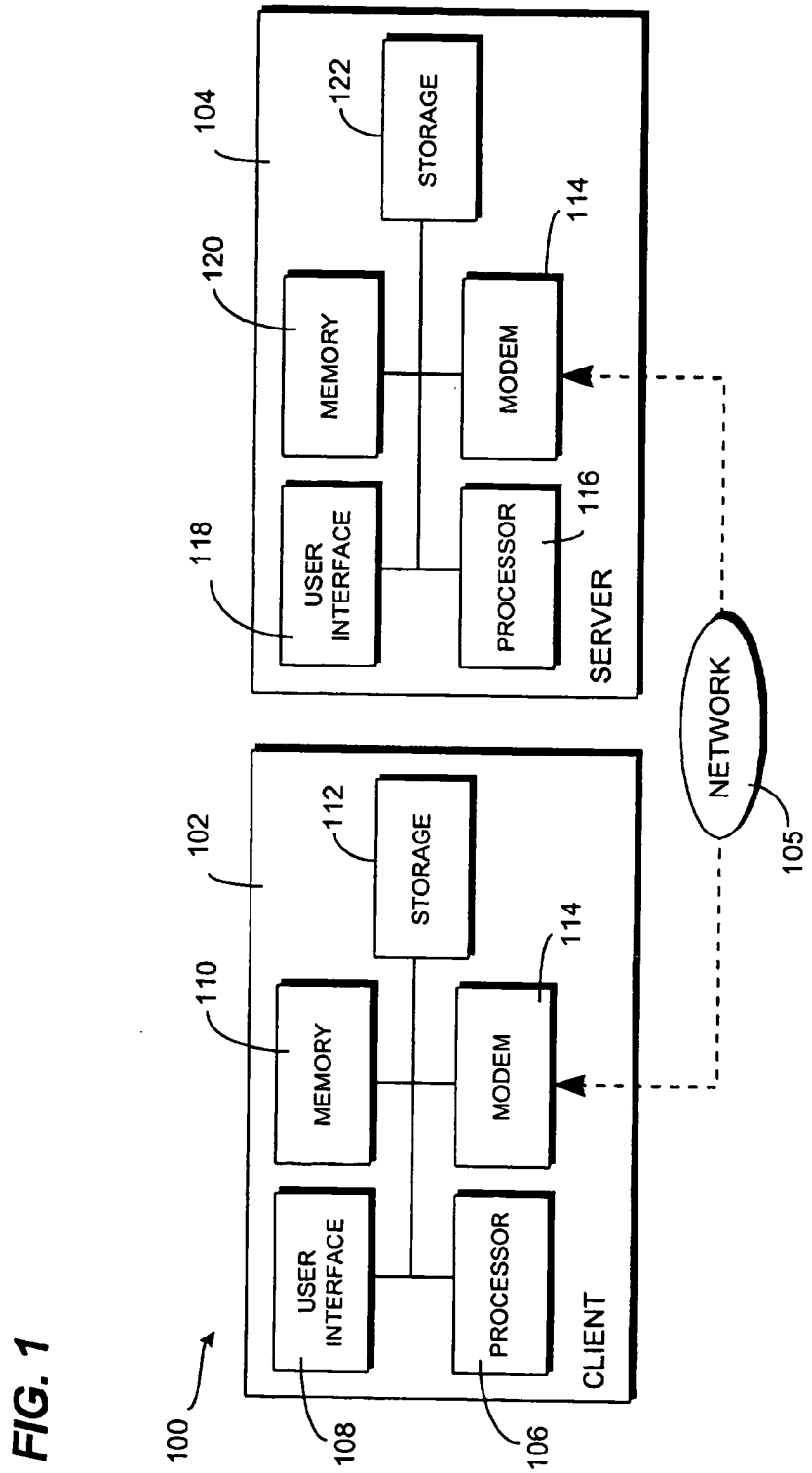

100
102
104
CLIENT
SERVER
NETWORK
105
106
108
110
112
114
116
118
120
122
STORAGE
MEMORY
MODEM
USER INTERFACE
PROCESSOR

FIG. 1

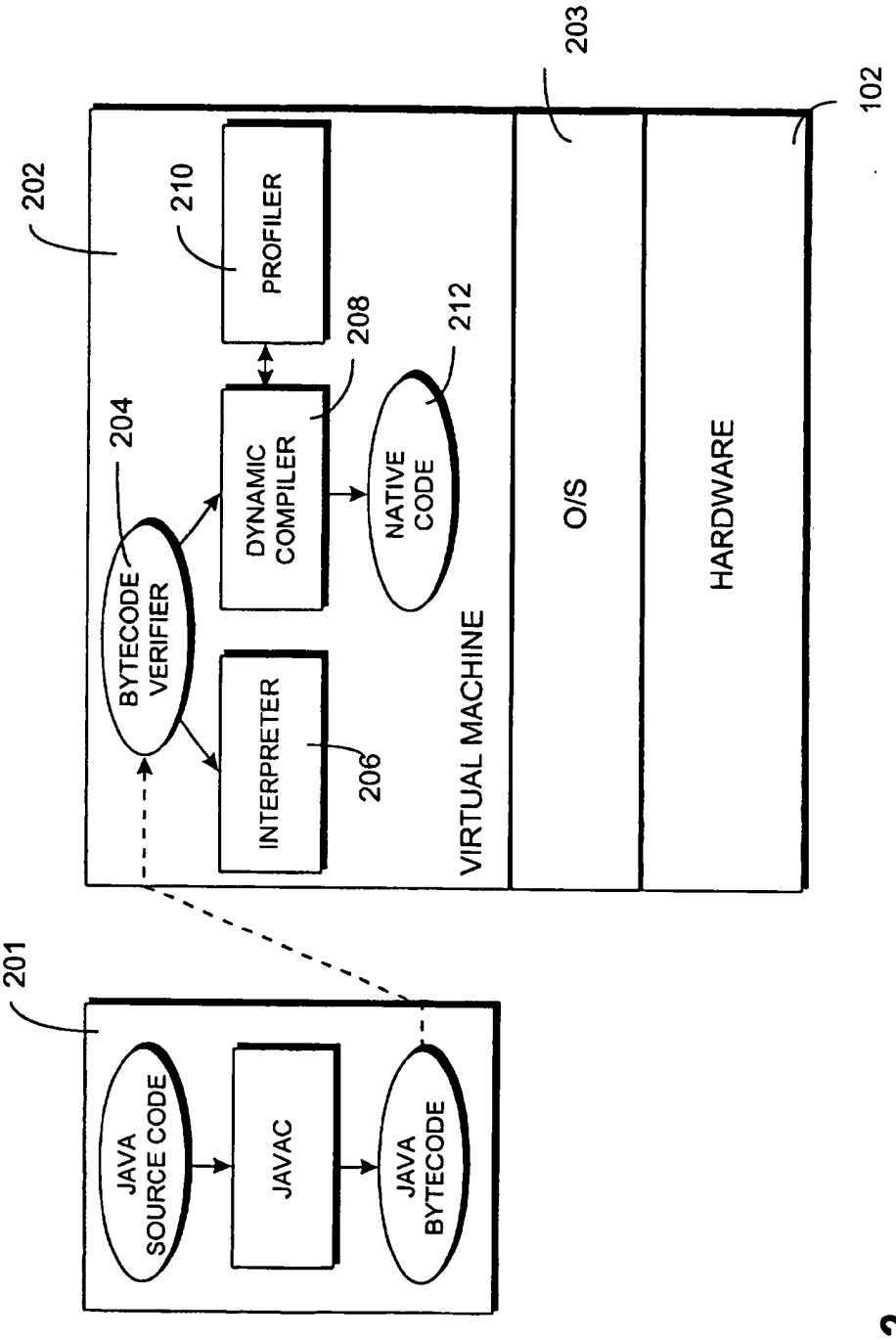

201
JAVA SOURCE CODE
JAVAC
JAVA BYTECODE
202
203
102
210
PROFILER
208
212
204
DYNAMIC COMPILER
NATIVE CODE
BYTECODE VERIFIER
INTERPRETER
206
VIRTUAL MACHINE
O/S
HARDWARE

FIG. 2

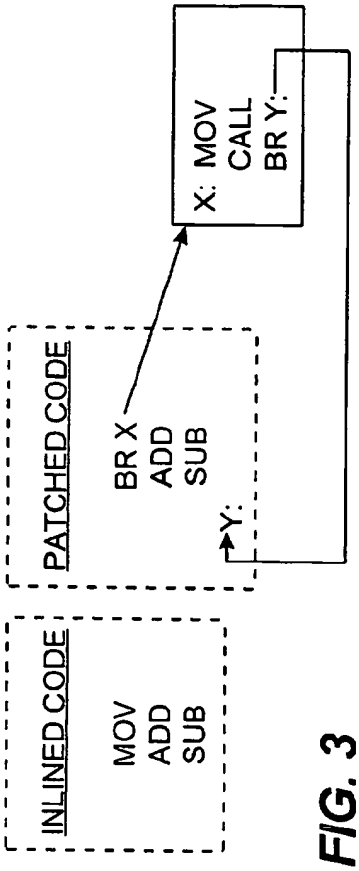

INLINED CODE
MOV
ADD
SUB
PATCHED CODE
BR X
ADD
SUB
Y:
X: MOV
CALL
BR Y:

FIG. 3

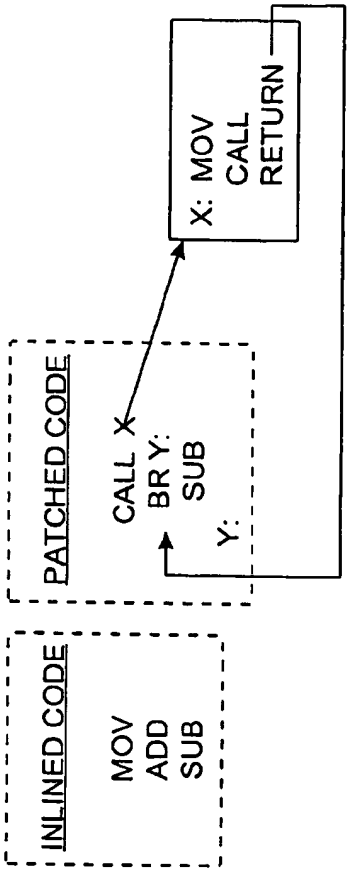

INLINED CODE
MOV
ADD
SUB
PATCHED CODE
CALL X
BR Y:
SUB
Y:
X: MOV
CALL
RETURN

FIG. 4

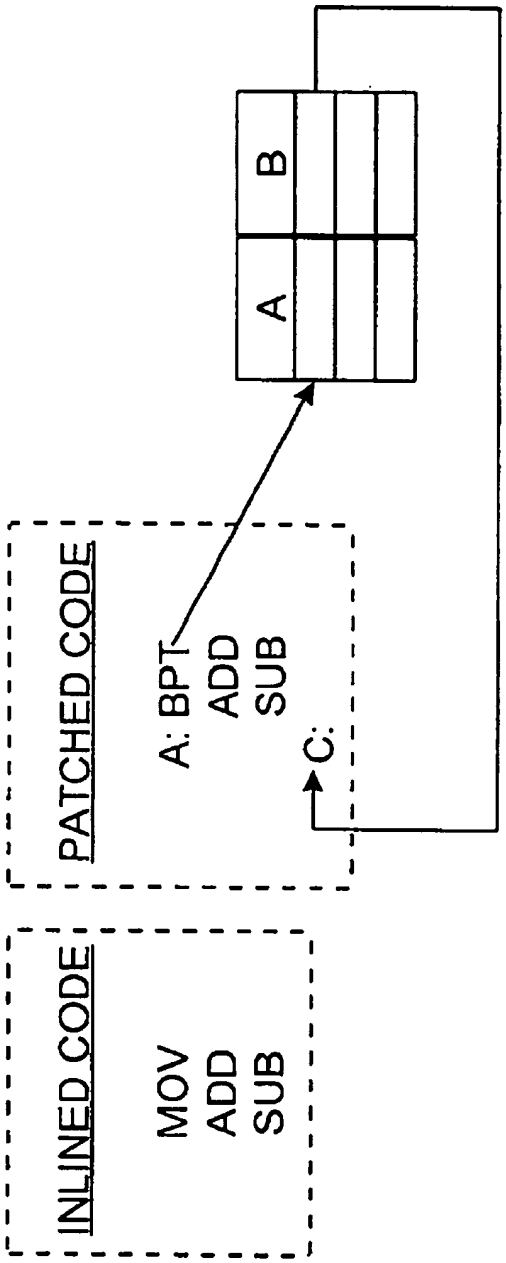


FIG. 5

CODE GENERATION FOR A BYTECODE COMPILER

BACKGROUND OF THE INVENTION

[0001] 1. Field of the Invention

[0002] The present invention relates, in general, to compilers, and, more particularly, to a code generation technique for optimizing execution of programs represented as bytecodes.

[0003] 2. Relevant Background

[0004] Bytecode programming languages such as the Java™ programming language (a trademark of Sun Microsystems, Inc.) represent computer programs as a set of bytecodes. Each bytecode is a numeric machine code for a "virtual machine" that exists only in software on a client computer. The virtual machine is essentially an interpreter that understands the bytecodes, interprets the bytecodes into machine code, then runs the machine code on a native machine.

[0005] Bytecode programming languages such as the Java programming language are gaining popularity among software application developers because they are easy to use and highly portable. Programs represented as bytecodes can be readily ported to any computer that has a virtual machine capable of properly interpreting and translating the bytecodes. However, because bytecode programming languages must be interpreted at run time on the client computer, they have suffered from an inability to execute at speed competitive with traditional compiled languages such as C or C++.

[0006] The speed limitations of bytecode languages are primarily related to the compilation process. Compilation is the process by which programs authored in high level language (i.e., human readable) are translated into machine readable code. There are four basic steps to the compilation process: tokenizing, parsing, code generation, and optimization. In traditional compiled programs all of these steps are completed prior to run time, whereas in interpreted languages such as BASIC, all of the compilation steps are performed at run time on an instruction-by-instruction basis. Command shells like CSH are also examples of interpreters that recognize a limited number of commands. Interpreted languages result in inefficiency because there is no way to optimize the resulting code.

[0007] In a bytecode programming language tokenizing and parsing occur prior to run time. After parsing the program is translated into bytecodes that can be interpreted by a virtual machine. As a result, a bytecode interpreter is faster than a language interpreter such as in some of the original BASIC programming language implementations. Also, the resulting programs when represented in bytecode format are more compact than a fully compiled program. These features make bytecode languages a useful compromise in networked computer environments where software is transferred from one machine for execution on another machine.

[0008] In the Java programming environment, the program that performs the translation to bytecodes is called "javac" and is sometimes referred to as a Java compiler (although it performs only part of the compilation process described above). The program that interprets the bytecodes on the client computer is called a Java virtual machine (JVM). Like other interpreters, the JVM runs in a loop executing each bytecode it receives. However, there is still a time consuming translation step on the virtual machine as the bytecodes are interpreted. For each bytecode, the interpreter identifies the corresponding series of machine instructions and then executes them. The overhead involved in any single translation is trivial, but overhead accumulates for every instruction that executes. In a large program, the overhead becomes significant compared to simply executing a series of fully-compiled instructions. As a result, large applications written in the Java programming language tend to be slower than the equivalent application in a fully compiled form.

[0009] To speed up execution, virtual machines have been coupled with or include a just-in-time compiler or JIT. The JIT improves run-time performance of bytecode interpreters by compiling the bytecodes into native machine code before executing them. The JIT translates a series of bytecodes into machine instructions the first time it sees them, and then executes the machine instructions instead of interpreting the bytecodes on subsequent invocations. The machine instructions are not saved anywhere except in memory, hence, the next time the program runs the JIT compilation process begins anew.

[0010] The result is that the bytecodes are still portable and in many cases run much faster than they would in a normal interpreter. Just-in-time compiling is particularly useful when code segments are executed repeatedly as in many computational programs. Just-in-time compiling offers little performance improvement and may actually slow performance for small code sections that are executed once or a few times and then not reused.

[0011] One limitation of JIT technology is that the compiling takes place at run-time and so any computation time spent in trying to optimize the machine code is overhead that may slow execution of the program. Hence, many optimization techniques are not practical in prior JIT compilers. Also, a JIT compiler does not see a large quantity of code at one time and so cannot optimize over a large quantity of code. One result of this is that the compiler cannot determine with certainty the set of classes used by a program. Moreover, the set of classes can change each time a given program is executed so the virtual machine compiler can never assume that the set of classes is ever unambiguously known.

[0012] Because of this uncertainty, it is difficult to produce truly optimal native code at run-time. Attempts to optimize with incomplete knowledge of the class set can be inefficient or may alter program functionality. Uncertainty about the class set gives rise to a large area of potential inefficiency in Java execution. The class set is a set of all classes that will be used to execute a particular instance of a Java language program. In a traditionally batch-compiled program, the entire universe of classes that will be used by the program is known at compile time greatly easing the optimization task.

[0013] A common program occurrence is that a first method (the caller method) calls a second method (the target method) in a process referred to as a "method call". This method call process is advantageous from a programmers perspective, but consumes many clock cycles. An important

optimization in the compilation of object oriented program code is called "inlining". In fully compiled programs, inlining makes these target method calls more efficient by copying the code of the target method into the calling method.

[0014] However, because of the semantics of the Java platform the compiler cannot ever determine the entire class set. Because classes are extensible and dynamically loadable in Java, methods can be overridden by subsequent extensions to a class, the compiler cannot know with certainty that a virtual method call will reach any particular method. In the Java programming language, a method by default can be overridden unless it is expressly defined as "final". All "non-final" methods are assumed to be overrideable.

[0015] Hence, all calls to non-final leaf methods must assume that the target method may be overridden at some time in the future. Hence, only time consuming virtual method call sequences have been used to invoke them in the past. The smaller the method, the more advantage inlining gives. A one-line method would typically spend far more time entering and exiting the routine as it does executing its contents. Tests have revealed that often as much as 85% of these non-final leaf method calls could be resolved if it were possible to know with certainty that no further classes would overload the methods.

[0016] What is needed is a method and apparatus for producing more optimal native code in an environment where the set of classes for a program is not unambiguously known. A need also exists for method and apparatus that deals with cases when knowledge of the class set is incomplete with tolerable impact on program execution performance.

SUMMARY OF THE INVENTION

[0017] Briefly stated, the present invention involves a method, system and apparatus for generating and optimizing native code in a runtime compiler from a group of bytecodes presented to the compiler. The compiler accesses information that indicates a likelihood that a class will be a particular type when accessed by the running program. Using the accessed information, the compiler selects a code generation method from a plurality of code generation methods. A code generator generates optimized native code according to the selected code generation method and stores the optimized native code in a code cache for reuse.

BRIEF DESCRIPTION OF THE DRAWINGS

[0018] **FIG. 1** illustrates a network computer environment implementing the method and system in accordance with the present invention;

[0019] **FIG. 2** shows a programming environment in accordance with the system and methods of the present invention; and

[0020] **FIG. 3** shows exemplary implementation of the present invention;

[0021] **FIG. 4** shows an alternative embodiment in accordance with the present invention; and

[0022] **FIG. 5** shows another alternative embodiment in accordance with the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0023] The present invention is based on a premise that a system that is used to repeatedly run the same bytecode language programs may build up a suitably accurate knowledge of the set of classes used in those programs through a process called "profiling". Profiling may occur over a single execution of the program or over a number of program executions that occur in different sessions. It is this accumulation of knowledge derived at run time that turns a limitation of prior systems, namely the lack of run time-specific knowledge, into an advantage. The advantage results because the accumulated knowledge is specific to the particular run time environment and inherently accounts for any idiosyncrasies of the run time environment when profiling the running programs. In contrast, conventional compilation processes optimize based on generalized knowledge of typical program behavior in typical run time environments.

[0024] The system in accordance with the present invention generates efficient native code from a bytecode program when it "knows" or can reasonably believe that the class set used in prior executions will not be expanded upon. The term "know" and "knowledge" as used herein means that data structures within the system and/or software hold data representing the information that is known. Similarly, a reasonable belief means that the system and/or software hold information that indicates a condition more likely than not exists even though that condition cannot be known with certainty.

[0025] The present invention generates code at run time using a dynamic compiler that is responsive to the information that known or reasonably believed about the program that is executing. In a particular example, the dynamic compiler includes a number of code generation methods that it can select from for any particular group of bytecodes that it is compiling. The selection of which method to use in a particular case is based upon the knowledge that can be gained instantaneously by probing the code itself together with knowledge gathered over time from monitoring the execution of code.

[0026] Although the present invention is described in terms of a system that compiles all bytecodes in a program to native code, it should be understood that the invention is usefully employed in a system that compiles only selected groups of bytecodes. For example, the virtual machine in accordance with the present invention may include a conventional bytecode interpreter that operates in parallel with the dynamic compiler. In such an implementation, only selected groups of bytecodes may be compiled based upon their frequency of execution, the relative benefit that may be achieved by compiling the bytecodes, or other performance based criteria. Accordingly, these implementations are equivalent to the specific implementations described herein for purposes of the present invention.

[0027] The present invention specifically addresses optimization of native code through heavy use of inlining techniques when the code involves target method calls. Hence, the present invention is particularly useful in optimizing object-oriented programs that make use of many small objects and methods that frequently make calls to methods from other objects and the classes that describe

those methods. The Java programming language is an example of a highly object oriented language that encourages tight object definitions using small limited purpose methods.

[0028] In a highly object oriented language, class extensibility is a key feature that eases programming and improves functionality. A class is extended when it is subclassed so that methods described by the class are overridden. Once overridden, all subsequent calls to an overridden method will use the new method, not the one originally described in the parent or abstract class when the receiver of the method call is the extended class. A significant utility of the present invention is that it provides a system and method for handling the complexities caused by subclassing in a running system.

[0029] **FIG. 1** illustrates a computer system **100** configured to implement the method and apparatus in accordance with the present invention. A client computer **102** downloads a program residing on a server computer **104**. The client computer has a processing unit **106** for executing program instructions that is coupled through a system bus to a user interface **108**. User interface **108** includes available devices to display information to a user (e.g., a CRT or LCD display) as well as devices to accept information form the user (e.g., a keyboard, mouse, and the like). A memory unit **110** (e.g., RAM, ROM, PROM and the like) stores data and instructions for program execution. Storage unit **112** comprises mass storage devices (e.g., hard disks, CDROM, network drives and the like). Modem **114** converts data from the system bus to and from a format suitable for transmission across network **105**. Modem **114** is equivalently substituted by a network adapter or other purely digital or mixed analog-digital adapter for a communications network.

[0030] Server **104** typically comprises a similar group of components including a processor **116**, a user interface **118**, and server memory **120**. Server storage **122**, in a particular example, stores programs represented in bytecode that are transmitted via modem **114** through network **105** to client computer **104**. It should be understood that the present invention may be implemented on single computer rather than the network computer environment shown in **FIG. 1** simply by storing the bytecode program on client storage unit **112**.

[0031] In the description of the present invention, the term "class type" refers to characteristic that identifies a particular class and distinguishes a class from any other class or any subclass of the class itself. When a class is subclassed, a new class type results. The class type of a particular object can be determined from data stored in the class description. A class may be defined using the "final" keyword in which case it cannot be subclassed, otherwise it can be subclassed. Object oriented programming encourages subclassing, so the usual case (and indeed the default case in the Java programming language) is that a class is subclassable. When a class is subclassable, it may be subclassed any number of times resulting in a set of class types created over the entire program execution. At runtime when a method is called through a method call, only one particular type of this set of types is correct.

[0032] **FIG. 2** illustrates an exemplary Java programming environment including a "compile time environment"**201** and a "run time environment"**202**. Developing and running a Java language program involves two steps. A programmer types in Java source code, which is converted by a Java compiler, such as the javac shown in **FIG. 2**, into a form called Java bytecode. As described above, the Java bytecodes are compact and portable which makes them an ideal form for storing and transferring a program in a network computer system.

[0033] The bytecode is then transferred to the runtime environment **202** to executed by a program known as the Java Virtual Machine (JVM). All JVMs understand the same bytecodes, so the bytecode form of a Java program can be run on any platform with a JVM. In this way, a JVM is a generic execution engine for Java bytecode—the JVM can be written once for a given platform, and then any bytecode program can be run by it. As in conventional JVMs, the preferred implementation includes a bytecode interpreter **206** that runs in parallel a dynamic compiler **208** to provide interpreted native code in a conventional manner while compiler **208** spends time generating optimized native code.

[0034] The dynamic compiler **208** in accordance with the present invention speeds up the execution of Java code by optimized compilation as described hereinbefore. Compiler **208** takes the Java bytecode and, in response to knowledge and performance information obtained from profiler **210**, converts the bytecodes into native code for the user's specific type of hardware **102** and operating system **203**. The present invention implements optimizing compiler technology to obtain high performance from the converted natively executable form of the program. Unlike the interpreted code from interpreter **206**, optimized code from compiler **208** is saved for later reuse in code cache **212**.

[0035] Profiler **210** is coupled to monitor code execution over time principally to monitor subclassing and overloading behavior of the particular program that is running. Of importance to the present invention, profiler **210** accumulates knowledge as to what classes are actually in use by the program. Profiler **210** generates profile data that is stored on a class-by-class or method-by-method basis in a profile data structure (not shown). Hence, although dynamic compiler **208** does not have any a priori knowledge of the class set, session-specific knowledge of the class set and class behavior can be obtained by profiler **210**. In an exemplary implementation, profiler **210** includes data structures such as a table, having an entry for each class that it sees executed.

[0036] As execution proceeds, profiler **210** stores profile data in the data structure indicating whether a particular class has been subclassed, and indicating whether one class type of a particular class is most likely to be the correct class type at any particular method invocation. The data stored is thus more useful than a simple "fixed" indication that can be determined directly from the class description. Moreover, while the data structure in profiler **210** can be initialized to some known state at startup, it is dynamically changing to reflect the current behavior of the program in the environment in which it is being used.

[0037] It is useful to understand the present invention in terms of four categories of method calls in which the invention is particularly useful. The first case is a method call when the receiver object's class (i.e., the class defining the target method) is unambiguously known. This may result when the method is indicated as "final" (i.e., includes the keyword final in its description) or fixed code analysis has

revealed the method to in fact be final even though it is not expressly declared final. In the second case the receiver class is known to be variable. A call to a method defined in an abstract class, or to a class known to be subclassed is an example of this case. These first two cases represent the extremes in which a class is known unambiguously to be fixed or the class is highly ambiguous with respect to which type it will be in a particular execution.

[0038] The remaining two cases involve gray areas in which some run time knowledge about the class can be exploited to provide generation of more efficient code. In the third case the receiver class is identified as one that more often than not is of a particular type. In accordance with the present invention, this case can be identified from profile-based feedback. In a fourth case the receiver class is very likely to be of a particular type because profile based feedback indicates that the class has never been subclassed in any previous execution.

[0039] In accordance with the present invention, in the first case it is known that the class cannot be overridden by subsequent method calls. In this case, method calls can always be called directly or inlined into the code generated for the calling method. These two types of optimizations (direct calls and inlining) are distinct but analogous operations. A direct call is faster to execute than a virtual method call. A still faster optimization is to actually copy the target method's code into the calling method's code (i.e., inlining). However, inlining will cause the code size to increase at run time and so place greater demands on the computer's instruction cache. Unless specified otherwise, the choice between these two optimization techniques is a matter of the appropriate code generation to meet the needs of a particular application.

[0040] In case two, a simple implementation of the present invention would generate code that implements a traditional virtual function call. This is justifiable because optimization requires greater overhead in such a case and may result in a very large amount of code to be generated for each call to the method. Alternatively, all possible classes are identified as cases in a "switch statement" so that the target method's code from each possible class is inlined into the code for the calling method. The switch is selected when the calling method is executed by testing the current class type and selecting the switch based on the test results.

[0041] In the third case, calls can be made directly to the method most likely to be correct with a small test being made in the code of the target method. If the class is not the correct one for the method, a virtual function call can be made. Alternatively, case three methods can be inlined, but with a test just before the first instruction of the inlined code that checks whether the class type is correct and makes a virtual method call instead if it is not.

[0042] In case four, method calls can be made efficient by calling the method directly. In accordance with the present invention, the code generated initially assumes that the class will never be subclassed. If, however, it subsequently is subclassed, then the first instruction of the method is patched to a small code fragment that tests the receiver and makes the virtual function call if it is not correct. If it is correct, it performs the instruction that was patched and branches back to the main method code.

[0043] The shorter the target method, the more important it is being inlined. To avoid recompilation of the calling methods, a test of the receiving class (i.e., the target method's class) will be required. Desirably this test adds little overhead and is relatively quick. A worst case example of a method calling a single instruction target method is demonstrated by a simple accessor method. For example assume a class Y having the description:

```
class Y {
    int value;              //declare "value" as an
integer                     //data type
    int length( ) {         //declare length( ) as a
method      return value;   //that returns
an integer "value"
    }                       //when called
}
    The calling method, for example, may be:
x = y.length( );
    The native code segment generated for the
calling method might become:
cmp [y+class],#constantForClassY      //compare y's
class type
                            //to a
specified constant
bne vfCall                  //branch when
the
                            //comparison
is not
                            //equal
mov[y+value],x              //move the
returned
                            //"value" to x
```

[0044] In the above machine code, the first instruction implements a class type test that compares a class type read from the class description in memory to a predefined constant for the expected class. This predefined constant is determined when the code is generated and can be updated whenever the code is regenerated. When the two values are equal, the branch-on-not-equal instruction is skipped and the machine code implementing the inline method is executed. The case for the call-through of the virtual function table is handled at vfCall and will result in some performance penalty. This is expected to be an unusual case and so the performance does not matter.

[0045] It is contemplated that the code could be organized so that results of this class type test could be reused for multiple method calls. In instances where several calls methods of the same class and the target method's class could not have been subclassed from one call to the other, a single class type test can be made and used by all of the target method calls. This implementation spread the overhead associated with performing the class type test over a plurality of method calls thereby improving efficiency.

[0046] A way to achieve further inlining improvement is to remove the class test (e.g., the CMP and BNE instructions in the above example) altogether. The resulting code includes only the target method's code and so is very efficient. However, the inlined code will not be accurate if the target method is ever overridden. Hence, when this is done, the JVM must remember that if the target method's class is ever subclassed by a class that overrides the method, that inlined code segment must be replaced with a virtual call sequence.

[0047] The virtual call sequence could be bigger than anything it is replacing, so to solve this, the first instruction

of the originally inlined code can be changed to an instruction that redirects program execution to a location at which instructions for implementing the virtual call sequence are located. Examples include a branch instruction, a call instruction, a breakpoint instruction, and the like. The choice of how to implement this feature is based on what gives a suitably efficient and flexible implementation for a particular machine.

[0048] In a branch implementation shown in **FIG. 3**, the first instruction of the inline code is replaced by a branch instruction (e.g., BR X where X is the location of the code implementing the virtual function call. On subsequent executions, the inlined code will not be executed and instead the code at location X is executed. The code implementing the virtual function call must end with a branch instruction to return to the end of the inlined code segment.

[0049] In a "call" implementation shown in **FIG. 4**, the first instruction of the inline code is replaced by a call instruction that call instructions at a location X that implement the virtual function call. At the end of the called sequence, a return instruction (RTN in **FIG. 4**) returns instruction execution to the location just after the call instruction in the patched code. The second instruction of the inline code is patched with a branch instruction pointing to the end of the inline code segment.

[0050] A call implementation shown in **FIG. 4** can be very efficient and flexible in some processors, such as SPARC architecture processors, that implement call instructions in a manner that enables the called instruction sequence to be anywhere in the virtual memory space of the computer (e.g., 4 gigabtyte address space in the case of SPARC architecture). In contrast, a typical branch instruction can only branch to a location within a limited range of address spaces around the branch instruction. This can be a significant advantage in cases where the patch code is actually compiled at run time. For example, if the code at location X in **FIG. 3** and **FIG. 4** were compiled only after it was determined that it was needed (i.e., after the method from which the inlined code originated was overridden), there may not be nearby empty memory space for holding the patching code. Hence, the patching code may have to be located outside the address range accessible through a branch instruction. Alternatively, the program can be filled with empty memory spaces that can be used for holding patching code as needed, however, this is a somewhat inefficient use of memory resources that may bloat the program and complicate managing the instruction cache.

[0051] **FIG. 5** illustrates an implementation using the breakpoint handler of the operating system to implement the patching code. As shown in **FIG. 5**, the first instruction of the inlined code is patched with a breakpoint instruction that throws an exception upon execution. The breakpoint instruction will be caught by the breakpoint handler which refers to a data structure such as a breakpoint table **601**. For every breakpoint instruction "A", table **601** maps to a set of instructions "B" for handling the breakpoint. The instructions at B can in turn implement the virtual function call. It should be understood that this procedure will be quite a slow sequence, but in application environments where this case occurs rarely, a net performance improvement will be achieved. These inefficiencies can be recorded in the profile and corrected the next time the task is run.

[0052] In order to enable this feature, when the compiler creates the method, it must create a code fragment that implements the virtual function call as well as a code fragment implementing the inlined code itself. This virtual function call code fragment is called by the breakpoint handler and must be increment the saved return address to cause the code execution to jump to the end of the now invalid inlined code, and then perform a return from interrupt sequence. The address of the memory location to be patched and the address of the memory location holding the code fragment are placed in a data structure that is associated with the method being inlined. Whenever this method is subclassed, then the patch to the instructions in instruction data structure is made and the breakpoint handler is informed of the association.

[0053] An attractive feature about the system and method in accordance with the present invention is that the code, once generated, never needs to be recompiled and that no difficult stack patching procedures are required. Although there is some increase in the compiler complexity and the size of data structure, an improvement in code execution efficiency is apparent.

[0054] Although the invention has been described and illustrated with a certain degree of particularity, it is understood that the present disclosure has been made only by way of example, and that numerous changes in the combination and arrangement of parts can be resorted to by those skilled in the art without departing from the spirit and scope of the invention, as hereinafter claimed.

We claim:

1. A process for generating code for a program represented as bytecodes comprising:

receiving a group of the bytecodes representing a method to be executed;

when the received bytecodes represent a calling method that calls another method, determining if the target method's class is known unambiguously;

when the target method's class is known unambiguously, generating code for the received bytecodes that inlines code from the target method into the code for the calling method to be executed; and

saving the generated code for subsequent executions of the calling method.

2. The process of claim 1 wherein when the target method's class is ambiguous, the method further comprises:

determining if the target method's class is one known to be subclassed;

when the target method's class is known to be subclassed, generating code for the received bytecodes that implements a virtual function call to the method.

3. The process of claim 1 wherein when the target method's class is ambiguous, the method further comprises:

determining if the target method's class is one known to be subclassed;

when the target method's class is known to be subclassed, determining if all possible subclasses for the target method's class is known;

when all possible subclasses for the target method's class are known, the method further comprises:

generating a code fragment for each possible subclass wherein each code fragment inlines code from the target method as defined by the associated subclass;

generating code for the received bytecodes that implements a switch having a case for each code fragment; and

generating code for the received bytecodes that selects the switch based upon the current subclass state of the target method's class at the time the calling method is executed.

4. The process of claim 1 wherein when the target method's class is ambiguous, the method further comprises:

identifying a set of classes that are possibly correct for the target method's class;

selecting one class from the set of possible classes based on knowledge that the selected class will more likely than not be correct for any execution of the calling class;

generating a first code fragment that inlines code from the selected class into the code for the calling method to be executed;

generating a second code fragment that implements a virtual function call without inlining code;

generating testing code that is operative before the first and second code fragments to test during each execution of the calling method whether the selected class is the correct class;

generating branching code that causes code execution to select either the first code fragment or the second code fragment in response to the testing code; and

combining the first code fragment, second code fragment, testing code and branching code to form the generated code for the received bytecodes.

5. The process of claim 1 wherein when the target method's class is ambiguous, the method further comprises:

generating a first code fragment for the received bytecodes that inlines code from the target method's class as if the target method's class were fixed;

generating a second code fragment that implements a virtual function call to the target method and returns program execution to the first code fragment;

creating a data structure comprising an entry for the target method, wherein the entry stores a first address of a first instruction in the first code fragment, and a second address of a first instruction of the second code fragment;

each time the target method is overridden, patching the first instruction of the first code fragment with a third code fragment that transfers program execution to the second code fragment.

6. The process of claim 5 wherein the third code fragment comprises a branch instruction and the second code fragment includes a branch instruction that returns program execution to the end of the first code fragment.

7. The process of claim 5 wherein the third code fragment comprises a call instruction and the patching further comprises replacing a second instruction in the first code fragment with a branch instruction that returns program execution to the end of the first code fragment.

8. The process of claim 5 wherein the third code fragment comprises a breakpoint instruction that invokes a breakpoint handler routine.

9. The process of claim 8 wherein the breakpoint handler routine includes code that that returns program execution to the end of the first code fragment.

10. The process of claim 1 wherein the step of determining if the target method's class is one known to be subclassed comprises examining a history of prior executions of the method to be executed.

11. An apparatus for generating code from received bytecodes comprising:

a compiler receiving the bytecodes;

a data structure within the compiler having a plurality of entries, wherein each entry corresponds to an object-oriented program class referred to by a unique group of the bytecodes;

a data record within each entry of the data structure indicating whether the corresponding class is likely to be subclassed;

a code generator within the compiler and coupled to access the data records, wherein for each group of bytecodes the code generator selects among a plurality of code generation options based upon a current value of the data record corresponding to the class referred to by the group of bytecodes; and

a native code cache having a plurality of entries where each entry corresponds to a specific group of bytecodes, each entry holding the generated native code associated with the specific group of bytecodes, wherein each first time a particular group of bytecodes is received by the compiler the group of bytecodes is passed to the code generator, and each subsequent time the particular group of bytecodes is received by the compiler the bytecodes are translated to selected code from the native code cache.

12. The apparatus of claim 11 further comprising:

a profiler coupled to monitor the compiler over a period of time to accumulate knowledge about the actual classes and subclasses that are invoked during the programs execution.

13. A system for run time compilation of object oriented program code, the program code including code that defines a plurality of object oriented classes, the system comprising:

a run time compiler that receives the program code, identifies calling methods that call a target method, and determine if the target method's class is unambiguously known;

a code generator within the compiler that inlines code from the target method into the code for the calling method to be executed; and

a code cache comprising data structures that store the generated code for subsequent executions of the calling method.

14. A computer system comprising:

memory for storing a bytecode program, the bytecode program including a sequence of bytecodes, where groups of the bytecodes represent methods to be executed including a calling method and a target method, wherein the calling method includes an operation that calls the target method;

a data processing unit for executing native code;

a code cache for storing native code representations of at least some of the methods;

a compiler operative on a first time that a specific group of bytecodes is received to translate the received group of bytecodes into a native code representation of the bytecodes and store the native code representation in the code cache;

a profiler coupled to the data processing unit, the profiler having data structures storing information about each method's class indicating a liklihood that that method's class will be subclassed;

optimizing code within the compiler coupled to the profiler's data structures for selecting a code generation method based upon the stored information.

15. The computer system of claim 14 wherein the profiler further comprises a monitor that monitors the execution of each method and records when each method's class is subclassed; and

a history accumulator that modifies the stored information in the profiler's data structure to indicate the recorded monitor information.

16. The computer system of claim 14 wherein the optimizer code further comprises:

code coupled to the profiler data structures identifying when a method's class is a unambiguously known;

code coupled to the profiler data structures identifying when a method's class is a dynamic type but is more likely than not of a particular type;

code coupled to the profiler data structures identifying when a method's class is dynamic but has never been subclassed in any recorded execution; and

and code coupled to the profiler data structures identifying when a method's class is dynamic and likely to be any one of a plurality of types at any particular execution.

17. The computer system of claim 16 wherein the compiler further comprises:

a compilation method responsive to the identification that a method's class is unambiguously known to inlines native code representing the method into the native code for any calling method when generating code for the calling method.

18. The computer system of claim 16 wherein the compiler further comprises a compilation method responsive to the identification that a method's class is a dynamic type but is more likely than not of a particular type to generate code that:

inlines native code representing the method from the class of the particular type identified as more likely than not correct;

implements a virtual function call to a current type of the method's class;

tests whether the class of the particular type identified as more likely than not is in fact correct; and

branching to either the inline native code or the virtual function call code based upon the test.

19. The computer system of claim 16 wherein the compiler further comprises:

a compilation method responsive to the identification that a method's class is a dynamic type but has never been subclassed in any recorded execution to generate a first code fragment that inlines native code representing the target method from the target method's class and a second code fragment that implements a virtual function call to the target method's class;

a first location in the code cache holding the first code fragment;

a second location in the code cache holding the second code fragment;

a data structure holding an address of the first location and an address of the second location;

code for detecting whether the target method's class has been subclassed;

code for patching a breakpoint instruction into the address of the first location in the code cache in response to detecting that the target method's class has been subclassed; and

a breakpoint handling routine that executes the code at the second location in the code cache and returns to an address at the end of the first location in the code cache.

20. A computer data signal embodied in a carrier wave coupled to a computer for generating code in the computer for a program represented as bytecodes comprising:

a first code portion comprising code configured to cause the computer to receive a group of the bytecodes representing a method to be executed;

a second code portion comprising code configured to cause the computer to determine whether that the target method's class is likely to be of a particular type at every execution of the target method;

a third code portion coupled to the second code portion comprising code configured to cause the computer to select a code generation strategy based upon the determination made by the second code portion; and

a fourth code portion coupled to the third code portion for generating code implementing the group of bytecodes according to the selected code generation strategy.

* * * * *